Lab_08_Regularisation.m

```
%% Machine Learning
% Lab 8: Regularized Linear Regression and Bias–Variance
% ——— Water overflow ———
%{
In the first half of the exercise, you will implement regularized linear
regression to predict the amount of water flowing out of a dam using the
change of water level in a reservoir. In the next half, you will go through
some diagnostics of debugging learning algorithms and examine the effects
of bias v.s. variance.
%}
%% Initialization
clear ; close all; clc

%% =========== Part 1: Loading and Visualizing Data =============
%  We start the exercise by first loading and visualizing the dataset.
%  The following code will load the dataset into your environment and plot
%  the data.
%

% Load Training Data
fprintf('Loading and Visualizing Data ...\n')

% Load from ex5data1:
% You will have X, y, Xval, yval, Xtest, ytest in your environment
load ('ex5data1.mat');

% m = Number of examples
m = size(X, 1);

% Plot training data
figure(1)
plot(X, y, 'rx', 'MarkerSize', 10, 'LineWidth', 1.5);
xlabel('Change in water level (x)');
ylabel('Water flowing out of the dam (y)');

fprintf('Program paused. Press enter to continue.\n');
pause;

%% =========== Part 2: Regularized Linear Regression Cost =============
%  You should now implement the cost function for regularized linear
%  regression.
%

w = [1 ; 1];
C = linearRegCostFunction([ones(m, 1) X], y, w, 1);

fprintf(['Cost at theta = [1 ; 1]: %f '...
         '\n(this value should be about 303.993192)\n'], C);

fprintf('Program paused. Press enter to continue.\n');
pause;

%% =========== Part 3: Regularized Linear Regression Gradient =============
%  You should now implement the gradient for regularized linear
%  regression.
%
```

```
w = [1 ; 1];
[C, grad] = linearRegCostFunction([ones(m, 1) X], y, w, 1);

fprintf(['Gradient at theta = [1 ; 1]:  [%f; %f] '...
         '\n(this value should be about [-15.303016; 598.250744])\n'], ...
         grad(1), grad(2));

fprintf('Program paused. Press enter to continue.\n');
pause;


%% =========== Part 4: Train Linear Regression =============
%  Once you have implemented the cost and gradient correctly, the
%  trainLinearReg function will use your cost function to train
%  regularized linear regression.
%
%  Write Up Note: The data is non-linear, so this will not give a great
%                 fit.
%

%  Train linear regression with lambda = 0
lambda = 0;
[w] = trainLinearReg([ones(m, 1) X], y, lambda);

%  Plot fit over the data
figure(1)
plot(X, y, 'rx', 'MarkerSize', 10, 'LineWidth', 1.5);
xlabel('Change in water level (x)');
ylabel('Water flowing out of the dam (y)');
hold on;
plot(X, [ones(m, 1) X]*w, '--', 'LineWidth', 2)
hold off;

fprintf('Program paused. Press enter to continue.\n');
pause;

%% =========== Part 5: Learning Curve for Linear Regression =============
%  Next, you should implement the learningCurve function.
%
%  Write Up Note: Since the model is underfitting the data, we expect to
%                 see a graph with "high bias"

lambda = 0;
[error_train, error_val] = ...
    learningCurve([ones(m, 1) X], y, ...
                  [ones(size(Xval, 1), 1) Xval], yval, ...
                  lambda);

figure(2)
plot(1:m, error_train, 1:m, error_val);
title('Learning curve for linear regression')
legend('Train', 'Cross Validation')
xlabel('Number of training examples')
ylabel('Error')
axis([0 13 0 150])

```

```
fprintf('# Training Examples\tTrain Error\tCross Validation Error\n');
for i = 1:m
    fprintf('  \t%d\t\t%f\t%f\n', i, error_train(i), error_val(i));
end

fprintf('Program paused. Press enter to continue.\n');
pause;

%% =========== Part 6: Feature Mapping for Polynomial Regression =============
%  One solution to this is to use polynomial regression. You should now
%  complete polyFeatures to map each example into its powers
%

p = 8;

% Map X onto Polynomial Features and Normalize
X_poly = polyFeatures(X, p);
[X_poly, mu, sigma] = featureNormalize(X_poly);  % Normalize
X_poly = [ones(m, 1), X_poly];                   % Add Ones

% Map X_poly_test and normalize (using mu and sigma)
X_poly_test = polyFeatures(Xtest, p);
X_poly_test = bsxfun(@minus, X_poly_test, mu);
X_poly_test = bsxfun(@rdivide, X_poly_test, sigma);
X_poly_test = [ones(size(X_poly_test, 1), 1), X_poly_test];         % Add Ones

% Map X_poly_val and normalize (using mu and sigma)
X_poly_val = polyFeatures(Xval, p);
X_poly_val = bsxfun(@minus, X_poly_val, mu);
X_poly_val = bsxfun(@rdivide, X_poly_val, sigma);
X_poly_val = [ones(size(X_poly_val, 1), 1), X_poly_val];           % Add Ones

fprintf('Normalized Training Example 1:\n');
fprintf('  %f  \n', X_poly(1, :));

fprintf('\nProgram paused. Press enter to continue.\n');
pause;

%% =========== Part 7: Learning Curve for Polynomial Regression =============
%  Now, you will get to experiment with polynomial regression with multiple
%  values of lambda. The code below runs polynomial regression with
%  lambda = 0. You should try running the code with different values of
%  lambda to see how the fit and learning curve change.

lambda = 0;
[w] = trainLinearReg(X_poly, y, lambda);

% Plot training data and fit
figure(3);
plot(X, y, 'rx', 'MarkerSize', 10, 'LineWidth', 1.5);
plotFit(min(X), max(X), mu, sigma, w, p);
xlabel('Change in water level (x)');
ylabel('Water flowing out of the dam (y)');
title (sprintf('Polynomial Regression Fit (lambda = %f)', lambda));

figure(4);
[error_train, error_val] = ...
```

```
    learningCurve(X_poly, y, X_poly_val, yval, lambda);
plot(1:m, error_train, 1:m, error_val);

title(sprintf('Polynomial Regression Learning Curve (lambda = %f)', lambda));
xlabel('Number of training examples')
ylabel('Error')
axis([0 13 0 100])
legend('Train', 'Cross Validation')

fprintf('Polynomial Regression (lambda = %f)\n\n', lambda);
fprintf('# Training Examples\tTrain Error\tCross Validation Error\n');
for i = 1:m
    fprintf('  \t%d\t\t%f\t%f\n', i, error_train(i), error_val(i));
end

fprintf('Program paused. Press enter to continue.\n');
pause;

%% =========== Part 8: Validation for Selecting Lambda ==============
%  You will now implement validationCurve to test various values of
%  lambda on a validation set. You will then use this to select the
%  "best" lambda value.
%

[lambda_vec, error_train, error_val] = ...
    validationCurve(X_poly, y, X_poly_val, yval);

%close all;
figure(5)
plot(lambda_vec, error_train, lambda_vec, error_val);
legend('Train', 'Cross Validation');
xlabel('lambda');
ylabel('Error');

fprintf('lambda\t\tTrain Error\tValidation Error\n');
for i = 1:length(lambda_vec)
        fprintf(' %f\t%f\t%f\n', ...
            lambda_vec(i), error_train(i), error_val(i));
end

fprintf('Program paused. Press enter to continue.\n');
pause;
```

linearRegCostFunction.m

```
function [C, grad] = linearRegCostFunction(X, y, w, lambda)
%LINEARREGCOSTFUNCTION Compute cost and gradient for regularized linear
%regression with multiple variables

% Initialize some useful values
m = length(y); % number of training examples

% You need to return the following variables correctly
C = 0;
grad = zeros(size(w));

w_0 = w;
w_0(1) = 0;
penalty = (lambda / (2*m))*sum(w_0.^2);
C = (1/(2*m))*(sum(((X*w)-y).^2))+penalty;

grad = (1/m)*sum(((X*w)-y).*X)+((lambda/m)*w_0');

grad = grad(:);

end
```

trainLinearReg.m

```
function [w] = trainLinearReg(X, y, lambda)
%TRAINLINEARREG Trains linear regression given a dataset (X, y) and a
%regularization parameter lambda

% Initialize weights
init_w = zeros(size(X, 2), 1);

% Create "short hand" for the cost function to be minimized
costFunction = @(t) linearRegCostFunction(X, y, t, lambda);

% Now, costFunction is a function that takes in only one argument
options = optimset('MaxIter', 200, 'GradObj', 'on');

% Minimize using fmincg
w = fmincg(costFunction, init_w, options);

end
```

learningCurve.m

```
function [error_train, error_val] = ...
    learningCurve(X, y, Xval, yval, lambda)
%LEARNINGCURVE Generates the train and cross validation set errors needed
%to plot a learning curve

% Number of training examples
m = size(X, 1);

% You need to return these values correctly
error_train = zeros(m, 1);
error_val   = zeros(m, 1);
```

```
12
13 for i= 1:m
14     w              = trainLinearReg(X(1:i,:), y(1:i), lambda);
15     error_train(i) = linearRegCostFunction(X(1:i,:)   , y(1:i)   , w, 0);
16     error_val(i)   = linearRegCostFunction(Xval, yval, w, 0);
17 end
18
19 end
```

polyFeatures.m

```
1  function [X_poly] = polyFeatures(X, p)
2  %POLYFEATURES Maps X (1D vector) into the p-th power
3
4  % You need to return the following variables correctly.
5  X_poly = zeros(numel(X), p);
6
7  for i=1:p
8    X_poly(:,i)=X.^i;
9  end
10
11 end
```

featureNormalize.m

```
1  function [X_norm, mu, sigma] = featureNormalize(X)
2  %FEATURENORMALIZE Normalizes the features in X
3  %{
4  bsxfun  Binary Singleton Expansion Function
5      C = bsxfun(FUNC,A,B) applies the element-by-element binary operation
6      specified by the function handle FUNC to arrays A and B, with implicit
7      expansion enabled.
8  %}
9  mu = mean(X);
10 X_norm = bsxfun(@minus, X, mu);
11
12 sigma = std(X_norm);
13 X_norm = bsxfun(@rdivide, X_norm, sigma);
14
15 end
```

validationCurve.m

```
function [lambda_vec, error_train, error_val] = ...
    validationCurve(X, y, Xval, yval)
%VALIDATIONCURVE Generate the train and validation errors needed to
%plot a validation curve that we can use to select lambda

% Selected values of lambda (you should not change this)
lambda_vec = [0 0.001 0.003 0.01 0.03 0.1 0.3 1 3 10]';

% You need to return these variables correctly.
error_train = zeros(length(lambda_vec), 1);
error_val = zeros(length(lambda_vec), 1);

for i = 1:length(lambda_vec)
  lambda = lambda_vec(i);
  theta = trainLinearReg(X, y, lambda);
  error_train(i) = linearRegCostFunction(X   , y   , theta, 0);
  error_val(i)   = linearRegCostFunction(Xval, yval, theta, 0);
end

end
```